CS422 - Programming Language Design

Functional Programming Homework

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

This homework assignment requires you to modify the provided definition(s) of FUN to incorporate some other interesting language features, or to change the semantics of existing ones.

For each problem, handle

- the K definitions of the features that you defined or modified (no need for fancy typing or absolute rigor for the K definitions, you can write them by hand); and
- the new Maude definition.

The former will be needed to understand your solution, and the latter for checking it against some programs.

Deadline: *October 31, midnight*.

Problem 1

In the current definition of FUN, there is no semantic boundaries between functions and control-intensive statements, such as exceptions or break/continue of loops.

In most cases this is desirable. For example, a language designer indeed wants exceptions to be potentially thrown from functions and to pass the functions' boundaries at no cost.

However, even though the break and continue of loops act as some sort of exceptions, one may find it inappropriate in one language design to allow break and continue pass the boundary of a function and act on the loop from within the function was called. For example, in our current definition of FUN, a program of the form

```
...
let f() = {break ; return 0}
in
  while (Cond) {
  ...
  f();
  ...
}
...
```

is "correct" and the call of the function in the while loop will result in breaking the loop.

While one may argue that this is actually an interesting capability of the language and so one may opt to keep it, you are required to actually "fix" this problem in FUN and disallow break and continue to cross the boundaries of functions.

You are *not* required to define a type checker or any static analyzer of the code in order to reject programs whose functions contain loop-unbounded break or continue constructs.

What you are required to do is to change the existing definition of FUN to "freeze" (no reductions can be applied anymore) when such a break or continue is encountered during an execution of a program.

Hint: empty the loop stack when calling a function (but make sure you recover it when returning from the function).

Modify the provided file `funk.maude`.

Problem 2

Define parametric break and continue.

More precisely, add to the FUN language constructs break(n) and continue(n) that skip over n loops when searching for a loop to break or continue. For example, the `break(3)` in the code below

```
while(C1) {
  while(C2) {
    while(C3) {
      while(C4) {
        break(3)
...
```

breaks the outmost loop `while(C1) {...}`. The current non-parametric break and continue become break(0) and `continue(0)`, respectively. Modify the file `funk.maude`.

Problem 3

"Fix" the semantics of *bind*.

Currently, *bind(Xl)* is defined to declare and properly bind as many locations as variables are in the list *Xl*, and, when a list of values *Vl* is passed to it in a continuation structure (i.e., "$Vl \curvearrowright bind(Xl)$"), to write those values at the corresponding locations. If there are fewer values in *Vl* than variables in *Xl*, then the remaining variables are not assigned values (but they are still allocated locations).

This (admittedly confusing) style of defining the semantics of *bind* allowed us to give the semantics of let, letrec and function invocations in a uniform way, using the same *bind* operator.

1. Give an example of a FUN program in which an "unintended" use of *bind*'s current generality is revealed; comment on whether this is a good or a bad feature to have in a language.
2. "Fix" this problem by defining two different *bind* operators, one for binding variables to values and another for binding variables to just locations; also, modify accordingly the definition of the current FUN language constructs that make use of binding.

Modify the file `funk.maude`.

Problem 4

Make FUN *dynamically scoped.*

FUN currently uses *static scoping* to detect what location a variable refers to. This means that the scope of a variable is determined statically, by analyzing the program before executing it. Most programmers tend to like static scoping, since it is easier to reason about programs written this way.

However, there exists another scoping approach, that of *dynamic scoping*, where the scope of a variable is determined at runtime by the current environment that the variable is being looked up.

Early versions of LISP were dynamically scoped; many say today that that happened "by mistake". UNIX's BC language is also dynamically scoped.

Although dynamic scoping works against the λ-calculus' invariance

to parameter names (aka α-equivalence) and is more difficult to analyze, it allows for supposedly easier and more efficient implementations; in particular, recursion is a non-issue in the context of dynamic binding.

Consider the following sample program:

```
let y = 1 in
  let f(x) = y in
    let y = 2 in
      f(0)
```

This evaluates to 1 under static scoping. However, it evaluates to 2 under dynamic scoping, since `y` is redefined before calling `f`.

Your task is to modify the definition of FUN in `funk.maude` to implement dynamic scoping. Basically, you need to "forget" about freezing the environment in closures ...

Problem 5

Add different parameter passing styles to FUN.

Let us briefly discuss the various parameter passing styles that you need to define.

Call-by-value. Under call-by-value, the argument expression is first evaluated and all its side effects are properly propagated; then the closure's body is evaluated in its environment extended with the binding of its parameter to the value obtained after evaluating the argument expression.

Call-by-reference. Call-by-reference makes sense only if the corresponding argument expression is a name. In that case, the formal parameter should be bound to the location of the argument.

Call-by-name. Under call-by-name, the corresponding argument expression is frozen and evaluated each time the parameter needs to

be evaluated in the function's body. The crucial aspect here is that, under static scoping, the expression is evaluated in the *environment in which the expression has been passed to the function*, not in the environment in which the function was declared.

Call-by-need. Call-by-need differs from call-by-name in what the frozen *expression is evaluated only once*, the first time its value is needed in order to evaluate the body of the function. Technically speaking, that means that the frozen expression is *unfrozen* and replaced by its value in the store at the appropriate location.

The side effects from evaluating the expression the first time have to be propagated as in the case of call-by-name. However, subsequent evaluations of that expression return directly the value obtained the first time it was evaluated, so the side effects are propagated *only once*.

Here is an example.

```
let val x = 1
in let ppstyle y = (x := x + x) ; x
   in x + y + y
```

where ppstyle $\in$ {val, name, need}.

If ppstyle = val then we have the usual semantics, that is, (x := x + x) ; x is first evaluated to 2 also changing the value of x, then x + y + y is evaluated to 6.

Under call by name, it is like replacing y by $(x := x + x); x$ in $x + y + y$, leading to $x + ((x := x + x); x) + ((x := x + x); x)$ which evaluates to $1 + 2 + 4 = 7$.

Finally, using call-by-need, y is unfrozen at first evaluation, becoming a value, so the result would be $1 + 2 + 2 = 5$.

The file `funk-param.maude` modifies the original `funk.maude` to make it more generic in what regards the various parametric passing styles. The following changes have been made:

- Syntax: when declaring a function or a let construct now variables have to be indexed by their passing style
- function semantics: since now actual parameters must be approached taking into account the style of the formal parameters, the semantics of function application becomes eager in the first argument. Only after evaluating the first argument to a closure we can proceed to evaluate and bind the other arguments according to their passing style.
- `PARAMETER-PASSING-STYLE` module adds common infrastructure for all passing styles.
- Pre-Values: we want to be able to store "frozen" expressions, but we do not want them to be dealt using the predefined

mechanism of accessing variables values – therefore a supersort of Val, PreVal is introduced.

Modify the provided file `funk-param.maude` to support all passing styles described above. Call by value is already provided. You are supposed to complete the commented-out definitions in modules `CALL-BY-NAME`, `CALL-BY-NEED` and `CALL-BY-REF`.

Note: you should not feel constrained to follow the given definitions: "just make it work."